Algorithm

I chose to code a version of the WALKSAT algorithm. This algorithm is based on the one presented in Russell and Norvig p226, copied here for simplicity:

```
function WALKSAT(clauses max-flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "randorn walk" move, typically around 0.5
          max-flips, number of flips allowed before giving up
model c a random assignment of truelfalse to the symbols in clauses
for i = 1 to max-flips do
          if model satisfies clauses then return model
          clause <-- a randomly selected clause from c1ause.s that is false in model
          with probability p flip the value in model of a randomly selected symbol from clause
          else flip whichever symbol in clause maximizes the number of satisfied clauses
return failure
```

The implementation is written in Python, instructions on how to run it are given in the last section.

The algorithm will operate on only CNF clauses and takes in the DIMACS format, a standard in SAT problems which is of the form:

c A sample .cnf file
p cnf **variables* *clauses**
v1 v2 v3 0
vn vm vj 0
…

where *variables* and *clauses* are the numeric value of the number of variables and clauses in the problem respectively, vi are the number of a specific variable in which a negative sign indicates a negated variable, and all lines end in a zero. The equivalent boolean expression would be:

$$(v_1 \vee v_2 \vee v_3) \wedge (v_n \vee v_m \vee v_j) \wedge ...$$

In order to optimize the algorithm I adjusted the two main parameters, the probability to flip a random variable and the maximum number of steps, to see how they affected performance. As one might expect, the harder a problem was, the more steps it took to solve. Thus maxSteps really only limits your ability to find a solution, False instances will always use maxSteps while true problems will hopefully take fewer. This variable was set to around 10,000 for the simpler problems so as not to limit the algorithm's ability to find solutions, while maintaining decent timing on False instances. pFlip was set to .5 as it gave a roughly 30% improvement over pFlip=.1.

Comparison

The SATLIB[1] sample problems are used to test the algorithm along with problems that were randomly generated by a program I created. The UBCSAT[2] package of algorithms was used as a comparison with optimized stochastic algorithms and ZChaff[3] was used as a comparison with an optimized DPLL algorithm.

Timing

My own sample problems were randomly generated by supplying the desired number of variables and

1 www.SATLIB.org
2 http://www.satlib.org/ubcsat/
3 http://www.princeton.edu/~chaff/zchaff.html

clauses, no further requirements were made. The SATLIB problems proved to be impossible for the algorithm I designed, though the two optimized algorithms were able to solve these 3-SAT problems in less than a second. The timing for my algorithm is shown in figure 1 and the timing on the professional algorithms is shown in figure 2.

Clearly these two professional algorithms are fully optimized and can solve much harder problems. Another issue may be the fact that my implementation is written in Python, which can run a factor of 2 slower or worse. Improvements to my own algorithm could be made by using a better heuristic (or a combination of heuristics).

It can be clearly seen that the stochastic algorithm runs much faster than the optimized DPLL algorithm (especially on hard problems), though this may be due to the specific choice in algorithm.

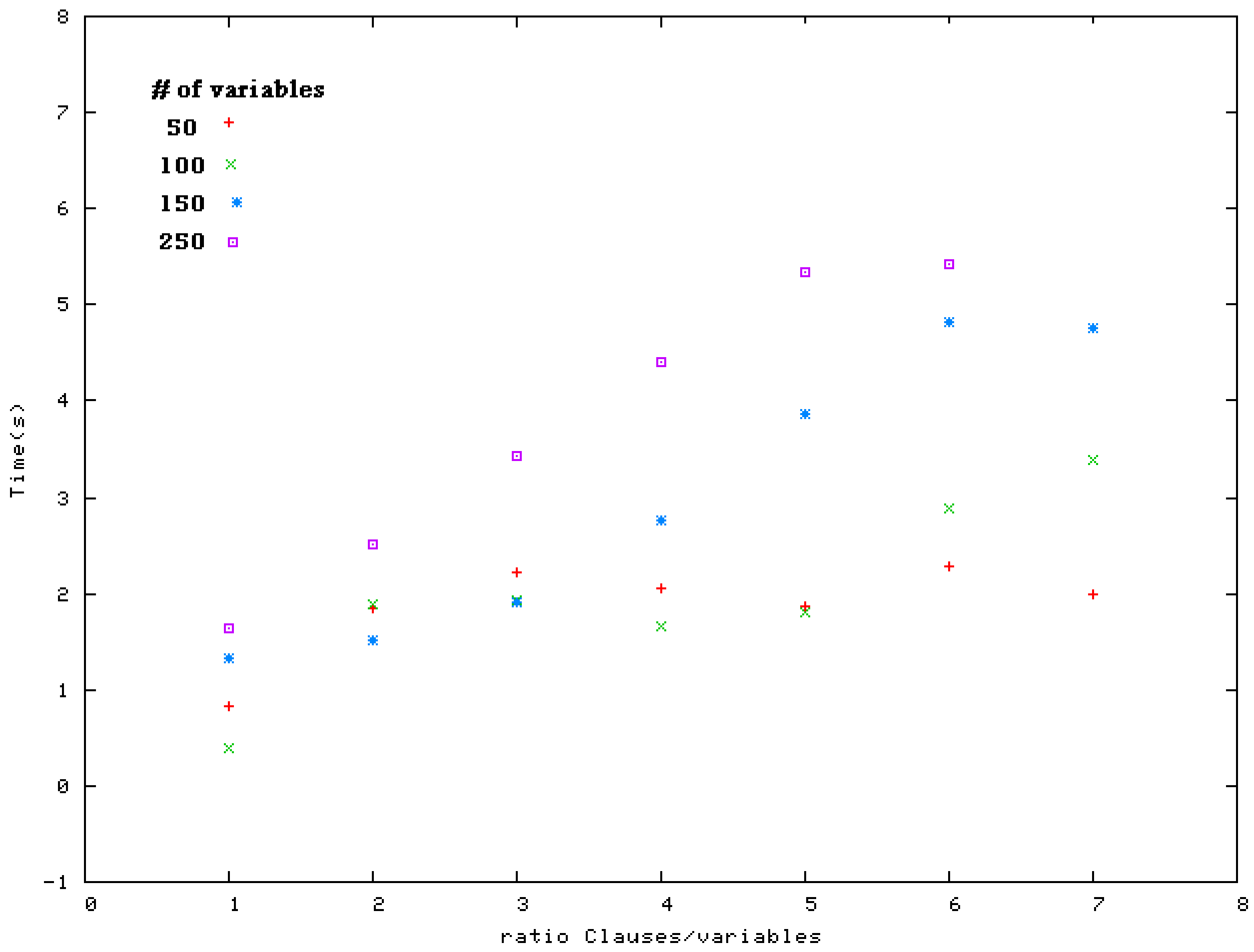

Illustration 1: The time taken to solve given randomized instances of a 3-SAT problem, averaged over 10 different random problems. Different number of variables and clause lengths were used and the ratio Clauses/Variables is plotted vs time.

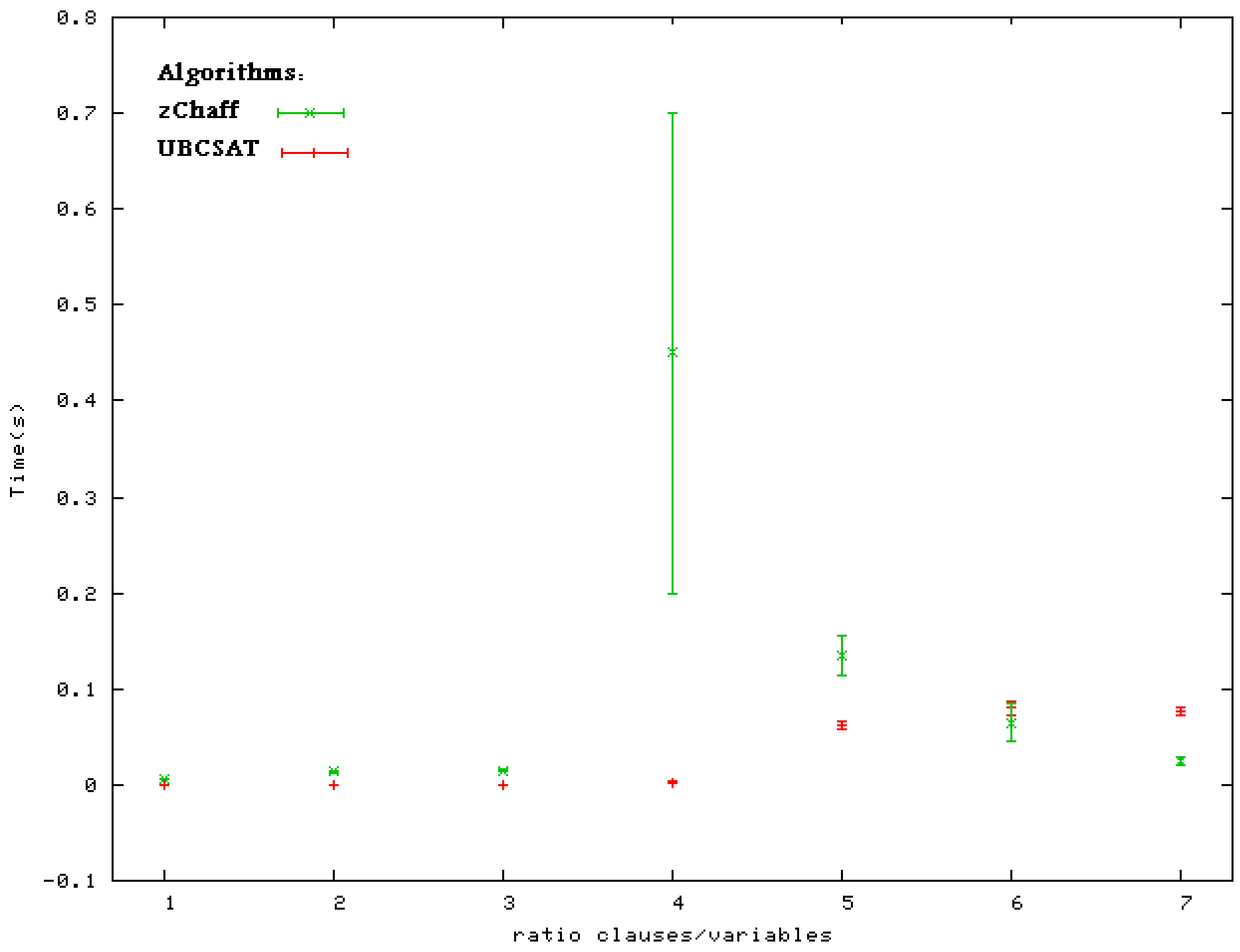

Illustration 2: The time taken to solve various random 3SAT problems for the two professional algorithms. The ratio of the number of variables to clauses is plotted versus the time, notice the huge time and standard deviation increase for the ratio around 4-5. The error bars represent the standard deviation in the time taken to solve various problems with that ratio.

With my algorithm, there was a large spread in time for various problems of a given number of clauses and variables, this reflects the varying difficulty between problems. Table 1 shows the standard deviation of the time taken for a sample of 10 different problems for each algorithm. The main point is that, though all of the algorithms see time and std increase with the clause/var ratio, my algorithm apparently suffers much more. This reflects that my Python WalkSAT implementation performs much worse on some problems while still performing reasonably on others.

clause/variables	*1*	*2*	*3*	*4*	*5*	*6*	*7*
zChaff	0.0005	0.0005	0.0003	0.25	0.01	0.01	0.002
UBCSAT	0.00005	0.00005	0.00005	0.005	0.002	0.002	0.003
Python WalkSAT	0.02	0.04	1.3	6	10	6	8

Table 1: The standard deviation of the time taken to solve a random instance of a 3sat problem as a function of the clause/var ratio. Notice that the std increases with the clause/var ratio and that the professional algorithms have a much lower std.

Another important factor to consider is the number of steps taken to solve the same problem during different executions of the algorithm. Figure 3 shows the time and number of steps taken to solve one instance of a simple problem (10 variables 40 clauses), notice that the variation in both time and steps is quite large. For comparison, the UBCSAT algorithm typically took 6 steps during execution, and (as shown before) was much faster. This is most likely due to a more optimized heuristic as well as other features.

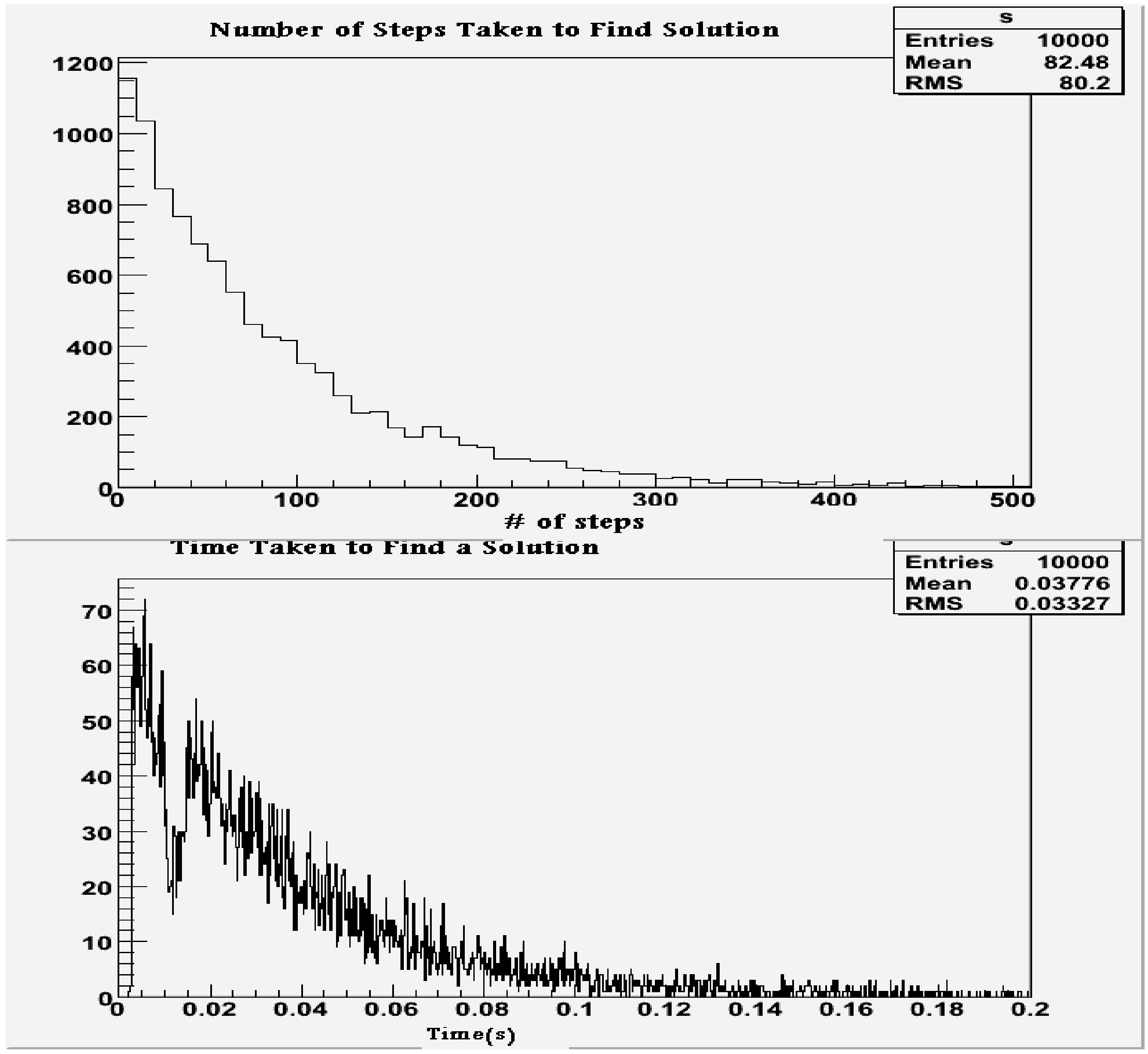

Illustration 3: One problem with 10 variables and 40 clauses was solved 10,000 times, the time and number of steps to find a solution are plotted above. For comparison, the average number of steps for the professional algorithm UBCSAT was 6 with a std of 0.8 (not plotted).

How to run the program

The program is written in Python v2.5 and can be executed by running: *python walkSAT.py* The program will automatically run on the included file sample.cnf and output the results (*time #Vars*

#Clauses truth) to the file results.dat. The input file can be changed in the main function or you can open up Python and type:

```
import walkSAT
walkSAT.walkSAT('filename')
```

which does not run the timing but just the walkSAT implementation.